

Streaming algorithms (extended abstract)

Jeremy Gibbons

Computing Laboratory, University of Oxford
www.comlab.ox.ac.uk/oucl/work/jeremy.gibbons
jeremy.gibbons@comlab.ox.ac.uk

Abstract. *Unfolds* generate data structures, and *folds* consume them. A *hylomorphism* is a fold after an unfold, generating then consuming a *virtual data structure*. A *metamorphism* is the opposite composition, an unfold after a fold; typically, it will convert from one data representation to another. In general, metamorphisms are less interesting than hylomorphisms: there is no automatic *fusion* to *deforest* the intermediate virtual data structure. However, under certain conditions fusion is possible: some of the work of the unfold can be done before all of the work of the fold is complete. This permits *streaming metamorphisms*, and among other things allows conversion of *infinite data representations*.

1 Introduction

Folds and *unfolds* in functional programming [14, 18, 1] are well-known tools in the programmer’s toolbox. Many programs that consume a data structure follow the pattern of a fold; and dually, many that produce a data structure do so as an unfold. In both cases, the structure of the program is determined by the structure of the data it processes.

It is natural to consider also compositions of these operations. A *hylomorphism* [20] consists of a fold after an unfold. The *virtual data structure* [24] produced by the unfold is subsequently consumed by the fold; the structure of that data determines the structure of both its producer and its consumer. Under certain rather weak conditions, the intermediate data structure may be eliminated or *deforested* [27], and the two phases fused into one slightly more efficient one.

In this paper, we consider the opposite composition, of an unfold after a fold. Programs of this form consume an input data structure using a fold, constructing some intermediate (possibly unstructured) data, and from this intermediary produce an output data structure using an unfold. Note that the two data structures may now be of different shapes, since they do not meet. Indeed, such programs may often be thought of as *representation changers*, converting from one structured representation of some abstract data to a different structured representation. We use the term *metamorphism* for such compositions.

In general, metamorphisms are perhaps less interesting than hylomorphisms, because there is no nearly-automatic deforestation. Nevertheless, sometimes fusion is possible; under certain conditions, some of the unfolding may be performed before all of the folding is complete. This kind of fusion can be helpful

for controlling the size of the intermediate data. Perhaps more importantly, it can allow conversions between infinite data representations. For this reason, we call such fused metamorphisms *streaming algorithms*; they are the main subject of this paper. We encountered them fortuitously while trying to describe some data compression algorithms [2], but have since realized that they are an interesting construction in their own right.

This paper is an extended abstract of a fuller version [13].

2 Notation

We are interested in capturing and studying *recurring patterns of computation*, such as folds and unfolds. As has been strongly argued by the recently popular *design patterns* movement [6], identifying and exploring such patterns has many benefits: reuse of abstractions, rendering ‘folk knowledge’ in a more accessible format, providing a common vocabulary of discourse, and so on. What distinguishes patterns in functional programming from patterns in object-oriented and other programming paradigms is that the better ‘glue’ available in the former [15] allows the patterns to be expressed as *abstractions within the language*, rather than having to resort to informal prose and diagrams.

We use the notation of Haskell [17], the de facto standard lazy functional programming language, except that we take the liberty to use some typographic effects in formatting, and to elide some awkwardnesses (such as type coercions and qualifications) that are necessary for programming but that obscure the points we are trying to make. (The true story, warts and all, is included in an appendix.)

This paper involves the datatype of lists:

```
data [α] = [] | α : [α]
```

That is, the datatype $[α]$ of lists with elements of type $α$ consists of the empty list $[]$, and non-empty lists of the form $a : x$ with head $a :: α$ and tail $x :: [α]$.

The primary patterns of computation over such lists are the *fold*, which consumes a list and produces some value:

```
foldr      :: (α → β → β) → β → [α] → β
foldr f b []    ≐ b
foldr f b (a : x) ≐ f a (foldr f b x)
```

and the *unfold*, which produces a list from some seed:

```
unfoldr    :: (β → Maybe (α, β)) → β → [α]
unfoldr f b ≐ case f b of
    Just (a, b') → a : unfoldr f b'
    Nothing     → []
```

Here, the non-recursive `Maybe` datatype is defined:

```
data Maybe α = Nothing | Just α
```


digits — the binary fraction must start with a zero. We make this intuition precise in Section 4; it involves, among other steps, inverting the structure of the traversal of the input by replacing the `foldr` with a `foldl`.

4 Streaming

As suggested in Section 3, it ought to be possible to produce some of the output in radix conversion before all of the input is consumed. In this section, we see how this can be done, developing our general results along the way.

4.1 The streaming theorem

The second phase of the metamorphism involves producing the output, maintaining some state in the process; that state is initialized to the result of folding the entire input, and evolves as the output is unfolded. Streaming must involve starting to unfold from an earlier state, the result of folding only some initial part of the input. Therefore, it is natural to consider metamorphisms in which the folding phase is an instance of `foldl`:

$$\text{unfoldr } f \cdot \text{foldl } g \ c$$

Essentially the problem is a matter of finding some kind of *invariant* of this state that determines the initial behaviour of the unfold. This idea is captured by the following definition.

Definition 1. *The streaming condition for f and g is:*

$$f \ c = \text{Just } (b, c') \Rightarrow f \ (g \ c \ a) = \text{Just } (b, g \ c' \ a)$$

for all a, b, c and c' .

Informally, the streaming condition states the following: if c is a state from which the unfold would produce some output element (rather than merely the empty list), then so is the modified state $g \ c \ a$ for any a ; moreover, the element b output from c is the same as that output from $g \ c \ a$, and the residual states c' and $g \ c' \ a$ stand in the same relation as the starting states c and $g \ c \ a$. In other words, ‘the next output produced’ is invariant under consuming another input.

This invariant property is sufficient for the unfold and the fold to be fused into a single process, which alternates between consuming inputs and producing outputs. We define:

$$\begin{aligned} \text{stream} & \quad :: (\gamma \rightarrow \text{Maybe } (\beta, \gamma)) \rightarrow (\gamma \rightarrow \alpha \rightarrow \gamma) \rightarrow \gamma \rightarrow [\alpha] \rightarrow [\beta] \\ \text{stream } f \ g \ c \ x & \hat{=} \mathbf{case} \ f \ c \ \mathbf{of} \\ & \quad \text{Just } (y, c') \rightarrow y : \text{stream } f \ g \ c' \ x \\ & \quad \text{Nothing} \quad \rightarrow \mathbf{case} \ x \ \mathbf{of} \\ & \quad \quad a : x' \rightarrow \text{stream } f \ g \ (g \ c \ a) \ x' \\ & \quad \quad [] \quad \rightarrow [] \end{aligned}$$

Informally, $stream\ f\ g :: \gamma \rightarrow [\alpha] \rightarrow [\beta]$ involves a producer f and a consumer g ; maintaining a state c , it consumes an input list x and produces an output list y . If f can produce an output element b from the state c , this output is delivered and the state revised accordingly. If f cannot, but there is an input a left, it is consumed and the state revised accordingly. When the state is wrung dry and the input is exhausted, the process terminates.

Formally, the relationship between the metamorphism and the streaming algorithm is given by the following theorem.

Theorem 2 (Streaming Theorem [2]). *If the streaming condition holds for f and g , then*

$$stream\ f\ g\ c\ x = unfoldr\ f\ (foldl\ g\ c\ x)$$

on finite lists x .

Proof. The proof is given in [2]. We prove a stronger theorem (Theorem 3) later.

Note that the result relates behaviours on finite lists only: on infinite lists, the `foldl` never yields a result, so the metamorphism may not either, whereas the streaming process can be productive — indeed, that is the whole point of introducing streaming.

4.2 Reversing the order of evaluation

In order to make a streaming version of radix conversion, we need to rewrite `fromBase` b as an instance of `foldl` rather than of `foldr`. There are standard techniques for doing this; Danvy [5] calls this transformation *defunctionalization*, and credits it to Reynolds [23], but it is also present in Wand’s seminal paper on continuation-based program transformations [28].

In the case of radix conversion, we get

$$fromBase\ b = app \cdot foldl\ (\otimes_b)\ ident$$

where

$$\begin{aligned} (n, c) \otimes_b m &\hat{=} (m + b \times n, b \times c) \\ ident &\hat{=} (0, 1) \\ app\ (n, b) &\hat{=} n \div b \end{aligned}$$

4.3 Checking the streaming condition

We cannot quite apply Theorem 2 yet, because the composition of `toBase` b' and the revised `fromBase` b has the abstraction function `app` between the `unfold` and the `fold`. Fortunately, that `app` fuses with the `unfold`. For brevity below, we define

$$unfoldr\ next_c \cdot app = unfoldr\ nextapp_c$$

where

$$\begin{aligned} \text{nextapp}_c(n, r) \hat{=} & \mathbf{if} \ n=0 \ \mathbf{then} \ \mathbf{Nothing} \ \mathbf{else} \ \mathbf{Just} \ (u, (n - u \times r \div c, r \div c)) \\ & \mathbf{where} \ u \hat{=} \lfloor n \times c \div r \rfloor \end{aligned}$$

Note that there was some leeway here: we had to partition the rational $n \times c \div r - u$ into a numerator and denominator, and we chose $(n - u \times r \div c, r \div c)$ out of the many ways of doing this. One might have expected $(n \times c - u \times r, r)$ instead; however, this leads to a dead-end, as we show later. Note that our choice involves generalizing from integer to rational components.

Having now massaged our radix conversion program into the correct format:

$$\text{radixConvert}(b, c) = \text{unfoldr nextapp}_c \cdot \text{foldl} (\otimes_b) \text{ident}$$

we may consider whether the streaming condition holds for nextapp_c and \otimes_b ; that is, whether

$$\begin{aligned} \text{nextapp}_c(n, r) &= \mathbf{Just} \ (u, (n', r')) \\ \Rightarrow \\ \text{nextapp}_c((n, r) \otimes_b m) &= \mathbf{Just} \ (u, (n', r') \otimes_b m) \end{aligned}$$

An element u is produced from a state (n, r) iff $n \neq 0$, in which case $u = \lfloor n \times c \div r \rfloor$. The modified state $(n, r) \otimes_b m$ evaluates to $(m + b \times n, b \times r)$. Since $n, b > 0$ and $m \geq 0$, this necessarily yields an element; this element v equals $\lfloor (m + b \times n) \times c \div (b \times r) \rfloor$. We have to check that u and v are equal. Sadly, they are generally not: since $0 \leq m < b$, it follows that v lies between u and $\lfloor (n + 1) \times c \div r \rfloor$, but these two bounds need not meet.

Intuitively, this can happen when the state has not completely determined the next output, and further inputs are needed in order to make a commitment to that output. For example, consider having consumed the first digit 6 while converting the sequence $[6, 7]$ in decimal (representing the fraction $67/100$) to ternary. The fraction $6/10$ is about 0.1210 in ternary; nevertheless, it is not safe to commit to producing the digit 1, because the true result is greater than 0.2_3 , and there is not enough information to decide whether to output a 1 or a 2 until the 7 has been consumed as well.

This is a common situation with streaming algorithms: the producer function (nextapp above) needs to be more cautious when interleaved with consumption steps than it does when all the input has been consumed. In the latter situation, there are no further inputs to invalidate a commitment made to an output; but in the former, a subsequent input might invalidate whatever output has been produced. The solution to this problem is to introduce a more sophisticated version of streaming, which proceeds more cautiously while input remains, but switches to the normal more aggressive mode if and when the input is exhausted. That is the subject of the next section.

4.4 Flushing streams

The typical solution is to introduce a ‘restriction’ safeapp of nextapp :

$$\text{safeapp} \ x \hat{=} \mathbf{if} \ \text{safe} \ x \ \mathbf{then} \ \text{nextapp} \ x \ \mathbf{else} \ \mathbf{Nothing}$$

for some predicate *safe*, and to use *safeapp* as the producer for the streaming process. In the case of radix conversion, the predicate *safe_c* (dependent on the output base *c*) could be defined

$$\text{safe}_c(n, r) \hat{=} (\lfloor n \times c \div r \rfloor = \lfloor (n+1) \times c \div r \rfloor)$$

That is, the state (n, r) is safe for the output base *c* if these lower and upper bounds on the next digit meet; with this proviso, the streaming condition holds, as we checked above. (In fact, we need to check not only that the same elements are produced from the unmodified and the modified state, but also that the two residual states are related in the same way as the two original states. With the definition of *nextapp_c* that we chose above, this second condition does hold; with the more obvious definition involving $(n \times c - u \times r, r)$ that we rejected, it does not.)

However, with this restricted producer the streaming process no longer has the same behaviour on finite lists as does the plain metamorphism: when the input is exhausted, the more cautious *safeapp* may have left some outputs still to be produced that the more aggressive *nextapp* would have emitted. The streaming process should therefore switch into a final ‘flushing’ phase when all the input has been consumed.

This insight is formalized in the following generalization of *stream*:

$$\begin{aligned} \text{fstream} &:: (\gamma \rightarrow \text{Maybe}(\gamma, \beta)) \rightarrow (\gamma \rightarrow \alpha \rightarrow \gamma) \rightarrow (\gamma \rightarrow [\beta]) \rightarrow \gamma \rightarrow [\alpha] \rightarrow [\beta] \\ \text{fstream } f \ g \ h \ c \ x &\hat{=} \text{case } f \ c \ \text{of} \\ &\quad \text{Just } (b, c') \rightarrow b : \text{fstream } f \ g \ h \ c' \ x \\ &\quad \text{Nothing} \quad \rightarrow \text{case } x \ \text{of} \\ &\quad \quad a : x' \rightarrow \text{fstream } f \ g \ h \ (g \ c \ a) \ x' \\ &\quad \quad [] \quad \rightarrow h \ c \end{aligned}$$

The difference between *fstream* and *stream* is that the former has an extra argument, *h*, a ‘flusher’; when the state is wrung as dry as it can be and the input is exhausted, the flusher is applied to squeeze the last few elements out. This is a generalization, because supplying the trivial flusher that always returns the empty list reduces *fstream* to *stream*. (In fact, *fstream* can also be expressed in terms of *stream*, so the two are equally expressive.)

The relationship of metamorphisms to flushing streams is a little more complicated than that to ordinary streams. One way of expressing the relationship is via a generalization of *unfoldr*, to yield not only the list of elements but also the final state, if there is one:

$$\begin{aligned} \text{evolve} &:: (\beta \rightarrow \text{Maybe}(\alpha, \beta)) \rightarrow \beta \rightarrow ([\alpha], \beta) \\ \text{evolve } f \ b &\hat{=} \text{case } f \ b \ \text{of} \\ &\quad \text{Just } (a, b') \rightarrow \text{let } (x, b'') \hat{=} \text{evolve } f \ b' \ \text{in } (a : x, b'') \\ &\quad \text{Nothing} \quad \rightarrow ([], b) \end{aligned}$$

Then $\text{unfoldr } f = \text{fst} \cdot \text{evolve } f$; if the generated list is infinite, the ‘final state’ is of course undefined. On finite inputs, provided that the streaming condition holds, a flushing stream process yields the same result as the ordinary streaming process, but with the results of flushing the final state (if any) appended.

Theorem 3 (Flushing Stream Theorem). *If the streaming condition holds for f and g , then*

$$fstream\ f\ g\ h\ c\ x = \mathbf{let}\ (y, c') \hat{=} evolve\ f\ (\mathbf{foldl}\ g\ c\ x)\ \mathbf{in}\ y \# h\ c'$$

on finite lists x .

Proof. See Appendix.

4.5 Radix conversion as a flushing stream

Returning for a final time to radix conversion, we define

$$safeapp_c(n, r) \hat{=} \mathbf{if}\ safe_c(n, r)\ \mathbf{then}\ nextapp_c(n, r)\ \mathbf{else}\ \mathbf{Nothing}$$

It is straightforward to prove that

$$\mathbf{unfoldr}\ nextapp_c\ nr = \mathbf{let}\ (y, nr'') \hat{=} evolve\ safeapp_c\ nr\ \mathbf{in}\ y \# \mathbf{unfoldr}\ nextapp_c\ nr'$$

We verified in Sections 4.3 and 4.4 that the streaming condition holds for $safeapp_c$ and \otimes_b . Theorem 3 then tells us that we can convert from base b to base c using

$$radixConvert(b, c) = fstream\ safeapp_c(\otimes_b)(\mathbf{unfoldr}\ nextapp_c)(0, 1)$$

This program works for finite or infinite inputs, and is always productive. (It does, however, always produce an infinite result, even when a finite result would be correct. For example, it will correctly convert $\frac{1}{3}$ from base 10 to base 2, but in converting from base 10 to base 3 it will produce an infinite tail of zeroes. One cannot really hope to do better, as returning a finite output depending on the entire infinite input is uncomputable.)

5 Related work

The notion of metamorphisms in general and of streaming algorithms in particular arose out of our work on arithmetic coding [2]. Since then, we have seen the same principles cropping up in other areas, most notably in the context of various kinds of numeric representations: the radix conversion problem from Section 3, continued fractions [13], and infinite compositions of homographies [12]. Potts [22] shows how to unify all these into a single framework; we intend to explore that unification. One might even see arithmetic coding as a kind of numeric representation problem.

Our theory of *metamorphisms* could easily be generalized to other datatypes: there is nothing to prevent consideration of folds consuming and unfolds producing datatypes other than lists. However, we do not currently have any convincing examples. However, it is not clear what a generic theory of *streaming algorithms* would look like. List-oriented streaming relies essentially on `foldl`, which does not generalize in any straightforward way to other datatypes. (We have attempted to show how to generalize `scanl` to arbitrary datatypes [7–9], and Pardo [21] has

improved on these attempts; but that work doesn't apply directly to folds as well as scans.)

Some of the ideas presented here appeared much earlier in work of Hutton and Meijer [16]. They study *representation changers*, consisting of a function followed by the converse of a function. Their representation changers are analogous to our metamorphisms, with the function corresponding to the fold and the converse of a function to the unfold: in a relational setting, an unfold is just the converse of a fold, and so our metamorphisms could be seen as a special case of representation changers in which both functions are folds. We feel that restricting attention to the special case of folds and unfolds is worthwhile, because we can capitalize on their universal properties; without this restriction, one has to resort to reasoning from first principles.

Acknowledgements

The idea of streaming algorithms came out of joint work with Richard Bird [2] and Barney Stratford. The principles behind reversing the order of evaluation and defunctionalization presented in Section 4.2 have been known for a long time [4, 28], but the presentation used here is due to Geraint Jones. The pattern of computation appearing in Theorem 3 is an instance of *primitive corecursion*, or what Vene and Uustalu [26] call *apomorphisms*.

We are grateful to members of the *Algebra of Programming* research group at Oxford and of *IFIP Working Group 2.1* and the participants in the *Datatype-Generic Programming* project for their helpful suggestions regarding this work.

References

1. Richard Bird and Oege de Moor. *The Algebra of Programming*. Prentice-Hall, 1996.
2. Richard Bird and Jeremy Gibbons. Arithmetic coding with folds and unfolds. In Johan Jeuring and Simon Peyton Jones, editors, *Advanced Functional Programming 4*, volume 2638 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.
3. Richard S. Bird. *Introduction to Functional Programming Using Haskell*. Prentice-Hall, 1998.
4. Eerke Boiten. The many disguises of accumulation. Technical Report 91-26, Department of Informatics, University of Nijmegen, December 1991.
5. Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In *Principles and Practice of Declarative Programming*, pages 162–174. SIGPLAN, 2001.
6. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
7. Jeremy Gibbons. *Algebras for Tree Algorithms*. D.Phil. thesis, Programming Research Group, Oxford University, 1991. Available as Technical Monograph PRG-94. ISBN 0-902928-72-4.
8. Jeremy Gibbons. Polytypic downwards accumulations. In Johan Jeuring, editor, *Proceedings of Mathematics of Program Construction*, volume 1422 of *Lecture Notes in Computer Science*, Marstrand, Sweden, June 1998. Springer-Verlag.

9. Jeremy Gibbons. Generic downwards accumulations. *Science of Computer Programming*, 37:37–65, 2000.
10. Jeremy Gibbons. Calculating functional programs. In Roland Backhouse, Roy Crole, and Jeremy Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *Lecture Notes in Computer Science*, pages 148–203. Springer-Verlag, 2002.
11. Jeremy Gibbons. Origami programming. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, Cornerstones in Computing. Palgrave, 2003.
12. Jeremy Gibbons. An unbounded spigot algorithm for the digits of π . Draft, November 2003.
13. Jeremy Gibbons. Metamorphisms and streaming algorithms. Unpublished draft, February 2004.
14. Tatsuya Hagino. A typed lambda calculus with categorical type constructors. In D. H. Pitt, A. Poigné, and D. E. Rydeheard, editors, *LNCS 283: Category Theory and Computer Science*, pages 140–157. Springer-Verlag, September 1987.
15. John Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, April 1989. Also in [25].
16. Graham Hutton and Erik Meijer. Back to basics: Deriving representation changers functionally. *Journal of Functional Programming*, 6(1):181–188, 1996.
17. Simon Peyton Jones. The Haskell 98 language. *Journal of Functional Programming*, 13, 2003.
18. Grant Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.
19. Lambert Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.
20. Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *LNCS 523: Functional Programming Languages and Computer Architecture*, pages 124–144. Springer-Verlag, 1991.
21. Alberto Pardo. Generic accumulations. In Jeremy Gibbons and Johan Jeuring, editors, *Generic Programming*, pages 49–78. Kluwer Academic Publishers, 2003. Proceedings of the IFIP TC2 Working Conference on Generic Programming, Schloß Dagstuhl, July 2002. ISBN 1-4020-7374-7.
22. Peter John Potts. *Exact Real Arithmetic using Möbius Transformations*. PhD thesis, Imperial College, London, July 1998.
23. John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher Order and Symbolic Computing*, 11(4):363–397, 1998. Reprinted from the Proceedings of the 25th ACM National Conference, 1972.
24. Doaitse Swierstra and Oege de Moor. Virtual data structures. In Bernhard Möller, Helmut Partsch, and Steve Schumann, editors, *LNCS 755: IFIP TC2/WG2.1 State-of-the-Art Report on Formal Program Development*, pages 355–371. Springer-Verlag, 1993.
25. David A. Turner, editor. *Research Topics in Functional Programming*. University of Texas at Austin, Addison-Wesley, 1990.
26. Varmo Vene and Tarmo Uustalu. Functional programming with apomorphisms (corecursion). *Proceedings of the Estonian Academy of Sciences: Physics, Mathematics*, 47(3):147–161, 1998. 9th Nordic Workshop on Programming Theory.
27. Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.
28. Mitchell Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180, January 1980.

A Proof of Theorem 3

The proof uses the following lemma [2], which lifts the streaming condition from single inputs to finite lists of inputs.

Lemma 4. *If the streaming condition holds for f and g , then*

$$f\ c = \text{Just}(b, c') \Rightarrow f(\text{foldl } g\ c\ x) = \text{Just}(b, \text{foldl } g\ c'\ x)$$

for all b, c, c' and finite lists x .

It also reduces in the case of an empty list of inputs to the following property, which can be proved using the *approximation lemma* [3, §9.3].

Lemma 5.

$$f\text{stream } f\ g\ h\ c\ [] = (\text{let } (y, c') \hat{=} \text{evolve } f\ c \text{ in } y \text{ ++ } h\ c')$$

Proof (of Theorem 3). Suppose that the streaming condition holds for f and g . We will prove by induction on x that, for all c and finite x ,

$$f\text{stream } f\ g\ h\ c\ x = \text{let } (y, c') \hat{=} \text{evolve } f(\text{foldl } g\ c\ x) \text{ in } y \text{ ++ } h\ c'$$

Case $x = []$. Then

$$\begin{aligned} & f\text{stream } f\ g\ h\ c\ [] \\ &= \{\text{Lemma 5}\} \\ & \text{let } (y, c') \hat{=} \text{evolve } f\ c \text{ in } y \text{ ++ } h\ c' \\ &= \{\text{foldl}\} \\ & \text{let } (y, c') \hat{=} \text{evolve } f(\text{foldl } g\ c\ []) \text{ in } y \text{ ++ } h\ c' \end{aligned}$$

Case $x = a : x'$. We make a case distinction on $f\ c$.

Subcase $f\ c = \text{Nothing}$. Then

$$\begin{aligned} & f\text{stream } f\ g\ h\ c\ (a : x') \\ &= \{f\text{stream}\} \\ & f\text{stream } f\ g\ h\ (g\ c\ a)\ x' \\ &= \{\text{induction}\} \\ & \text{let } (y, c') \hat{=} \text{evolve } f(\text{foldl } g\ (g\ c\ a)\ x') \text{ in } y \text{ ++ } h\ c' \\ &= \{\text{foldl}\} \\ & \text{let } (y, c') \hat{=} \text{evolve } f(\text{foldl } g\ c\ (a : x')) \text{ in } y \text{ ++ } h\ c' \end{aligned}$$

Subcase $f\ c = \text{Just}(b, d)$. Then Lemma 4 gives us

$$f(\text{foldl } g\ c\ x) = \text{Just}(b, \text{foldl } g\ d\ x)$$

and we have:

$$\begin{aligned}
& \mathbf{let} (y, c') \hat{=} \mathit{evolve} f (\mathit{foldl} g c x) \mathbf{in} y \mathbin{++} h c' \\
= & \quad \{\text{Lemma 4, } \mathit{evolve}\} \\
& \mathbf{let} (y, c') \hat{=} (\mathbf{let} (y', c'') \hat{=} \mathit{evolve} f (\mathit{foldl} g d x) \mathbf{in} (b : y', c'')) \mathbf{in} \\
& \quad y \mathbin{++} h c' \\
= & \quad \{\text{reassociating } \mathbf{lets}\} \\
& \mathbf{let} (y', c'') \hat{=} \mathit{evolve} f (\mathit{foldl} g d x) \mathbf{in} \\
& \quad \mathbf{let} (y, c') \hat{=} (b : y', c'') \mathbf{in} y \mathbin{++} h c' \\
= & \quad \{\text{expanding inner } \mathbf{let}\} \\
& \mathbf{let} (y', c'') \hat{=} \mathit{evolve} f (\mathit{foldl} g d x) \mathbf{in} b : y' \mathbin{++} h c'' \\
= & \quad \{\text{extracting } b \text{ from } \mathbf{let}\} \\
& b : \mathbf{let} (y', c'') \hat{=} \mathit{evolve} f (\mathit{foldl} g d x) \mathbf{in} y' \mathbin{++} h c'' \\
= & \quad \{\text{induction}\} \\
& b : \mathit{fstream} f g h d x
\end{aligned}$$